Name : VEDASHREE BHALERAO

Class : LY – AIA – 1

Batch : B

Roll No : 2213191

Assignment No. 6

Aim: Apply transfer learning with pre-trained VGG16 model on assignment 3 and analyze the result.

Objectives:

1. To learn pre-trained models

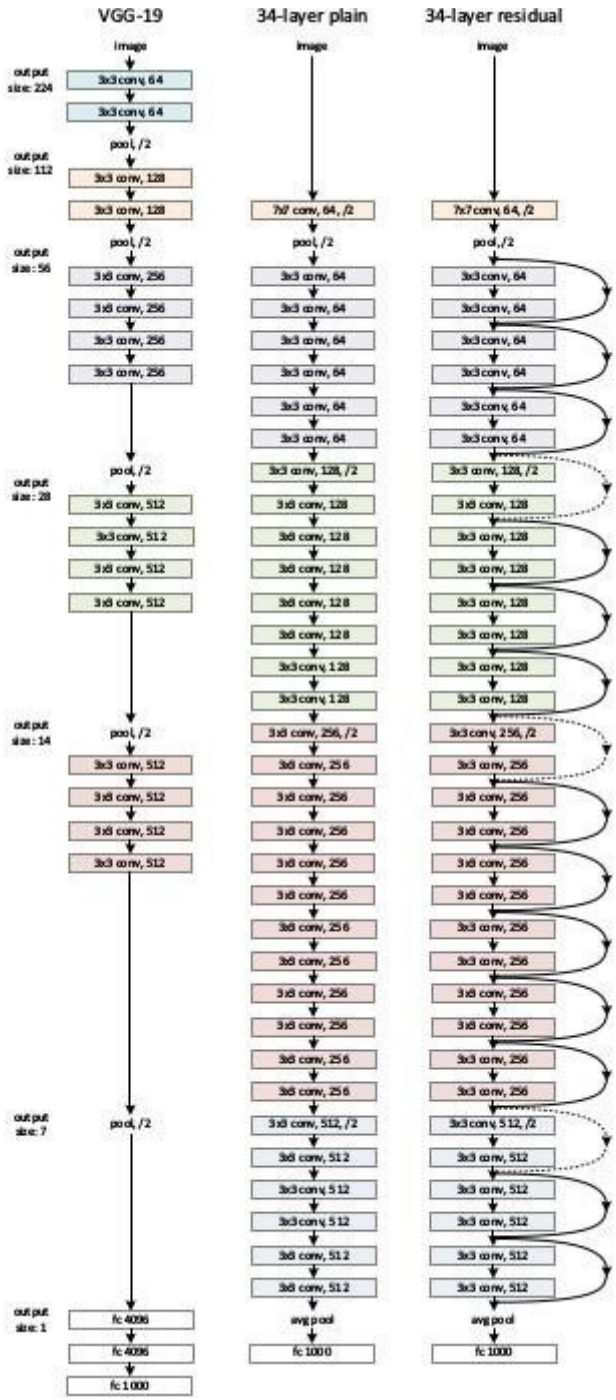
2. To learn transfer learning **Theory:**

Transfer learning

Transfer learning (TL) is a research problem in machine learning (ML) that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks. This area of research bears some relation to the long history of psychological literature on transfer of learning, although formal ties between the two fields are limited. From the practical standpoint, reusing or transferring information from previously learned tasks for the learning of new tasks has the potential to significantly improve the sample efficiency of a reinforcement learning agent.

ResNet 50

- Use 3*3 filters mostly.
- Down sampling with CNN layers with stride 2.
- Global average pooling layer and a 1000-way fully-connected layer with Softmax in the end.



Plain VGG and VGG with Residual Blocks

There are two kinds of residual connections:

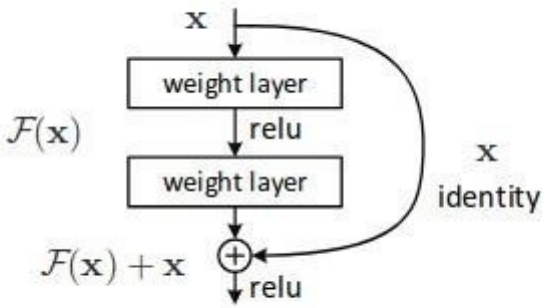


Figure 2. Residual learning: a building block.

Residual block

1. The identity shortcuts (x) can be directly used when the input and output are of the same dimensions.

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}. \qquad (1)$$

Residual block function when input and output dimensions are same

2. When the dimensions change, A) The shortcut still performs identity mapping, with extra zero entries padded with the increased dimension. B) The projection shortcut is used to match the dimension (done by 1*1 conv) using the following formula

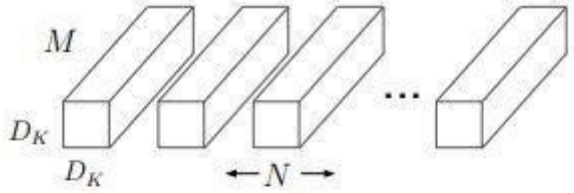
$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}. \qquad (2)$$

Residual block function when the input and output dimensions are not same.

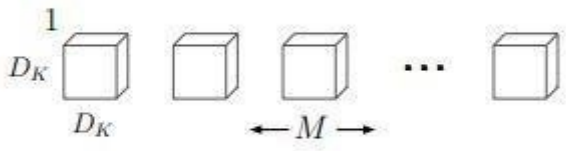
The first case adds no extra parameters, the second one adds in the form of W_{s}

MOBILENET

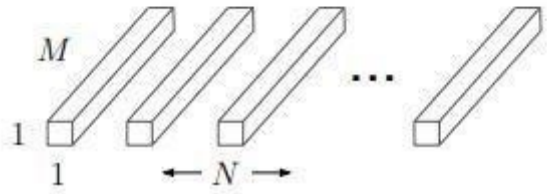
MobileNet is an efficient and portable CNN architecture that is used in real world applications. MobileNets primarily use **depthwise seperable convolutions** in place of the standard convolutions used in earlier architectures to build lighter models.MobileNets introduce two new global hyperparameters(width multiplier and resolution multiplier) that allow model developers to trade off **latency** or **accuracy** for speed and low size depending on their requirements.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



Architecture

MobileNets are built on depthwise seperable convolution layers.Each depthwise seperable convolution layer consists of a depthwise convolution and a pointwise convolution.Counting depthwise and pointwise convolutions as seperate layers, a MobileNet has 28 layers.A standard MobileNet has 4.2 million parameters which can be further reduced by tuning the width multiplier hyperparameter appropriately.
The size of the input image is 224 × 224 × 3.

The detailed architecture of a MobileNet is given below :

TYPE	STRIDE	KERNEL SHAPE	INPUT SIZE
Conv	2	3 × 3 × 3 × 32	224 × 224 × 3
Conv dw	1	3 × 3 × 32	112 × 112 × 32
Conv	1	1 × 1 × 32 × 64	112 × 112 × 32

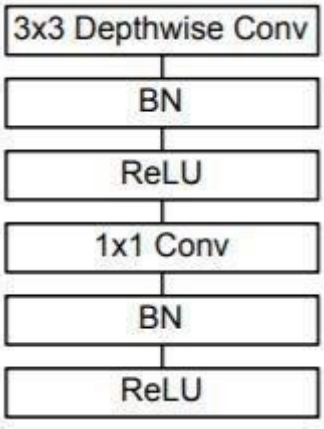
Conv dw	2	3 × 3 × 64	112 × 112 × 64
Conv	1	1 × 1 × 64 × 128	56 × 56 × 64
TYPE	**STRIDE**	**KERNEL SHAPE**	**INPUT SIZE**
Conv dw	1	3 × 3 × 128	56 × 56 × 128
Conv	1	1 × 1 × 128 × 128	56 × 56 × 128
Conv dw	2	3 × 3 × 128	56 × 56 × 128
Conv	1	1 × 1 × 128 × 256	56 × 56 × 128
Conv dw	1	3 × 3 × 256	28 × 28 × 256
Conv	1	1 × 1 × 256 × 256	28 × 28 × 256
Conv dw	2	3 × 3 × 256	28 × 28 × 256
Conv	1	1 × 1 × 256 × 512	14 × 14 × 256
Conv dw	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw	1	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 512	14 × 14 × 512
Conv dw	2	3 × 3 × 512	14 × 14 × 512
Conv	1	1 × 1 × 512 × 1024	7 × 7 × 512
Conv dw	2	3 × 3 × 1024	7 × 7 × 1024
Conv	1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Avg Pool	1	Pool 7 × 7	7 × 7 × 1024
Fully Connected	1	1024 × 1000	1 × 1 × 1024
Softmax	1	Classifier	1 × 1 × 1000

Standard Convoltion layer :

A single standard convolution unit (denoted by **Conv** in the table above) looks like this :

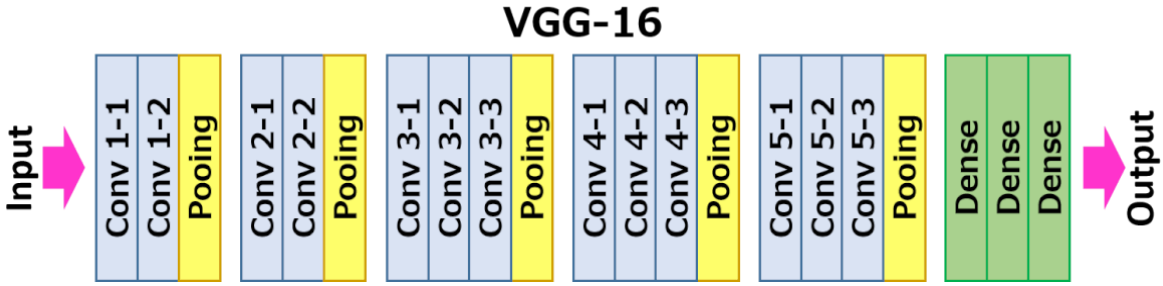
Depthwise seperable Convolution layer

A single Depthwise seperable convolution unit (denoted by **Conv dw** in the table above) looks like this :

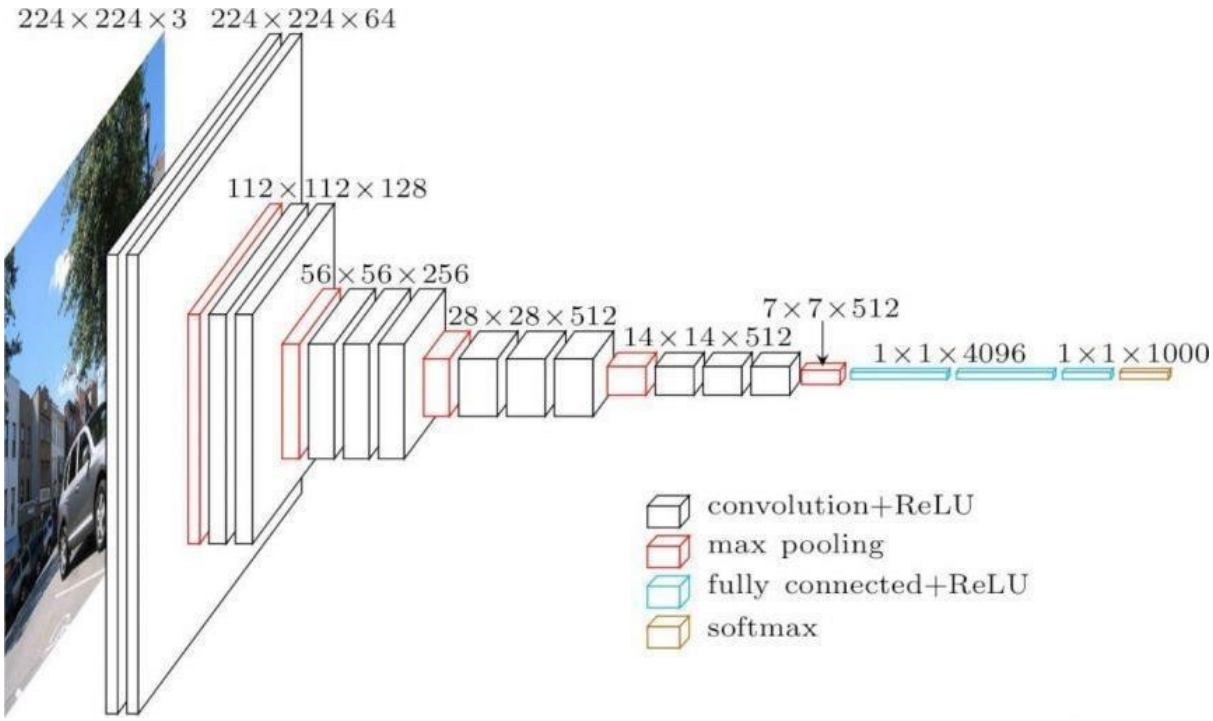


VGG16

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the
University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to ILSVRC2014. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU’s.



The architecture depicted below is VGG16.



VGG16 Architecture

The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the softmax layer. The configuration of the fully connected layers is the same in all networks.

Code:

```
import tensorflow as tf

from tensorflow.keras.applications import VGG16 from
tensorflow.keras.layers import Dense, Flatten, Dropout from
tensorflow.keras.models import Model from
```

```
tensorflow.keras.optimizers import Adam from
tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load and preprocess CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values to range 0-1 x_train
= x_train.astype('float32') / 255.0 x_test =
x_test.astype('float32') / 255.0

# Convert labels to one-hot encoding y_train
= to_categorical(y_train, 10) y_test =
to_categorical(y_test, 10)

# Load the VGG16 model without the top fully connected layers base_model =
VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
# Freeze all the layers in the base model
for    layer    in    base_model.layers:
layer.trainable = False

# Add custom layers on top of the base model x =
Flatten()(base_model.output) x = Dense(256,
activation='relu')(x) x = Dropout(0.5)(x)  # Add
dropout for regularization x = Dense(128,
activation='relu')(x) x = Dropout(0.5)(x) output =
Dense(10, activation='softmax')(x)

# Create the new model model =
Model(inputs=base_model.input, outputs=output)
# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
loss='categorical_crossentropy', metrics=['accuracy'])
# Train the model history = model.fit(x_train, y_train,
epochs=10, batch_size=64, validation_data=(x_test, y_test))

# Evaluate the model on test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc * 100:.2f}%")
# Plot training history import
matplotlib.pyplot as plt
  plt.plot(history.history['accuracy'], label='Train
Accuracy') plt.plot(history.history['val_accuracy'],
label='Val Accuracy') plt.xlabel('Epochs')
plt.ylabel('Accuracy') plt.legend() plt.show()
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.xlabel('Epochs') plt.ylabel('Loss') plt.legend()
plt.show()
```

Results:

```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 153s 193ms/step - accuracy: 0.1656 - loss: 2.2899 - val_accuracy: 0.4158 - va
l_loss: 1.7431
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 137s 176ms/step - accuracy: 0.3503 - loss: 1.8248 - val_accuracy: 0.4777 - va
l_loss: 1.5285
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 142s 182ms/step - accuracy: 0.4184 - loss: 1.6451 - val_accuracy: 0.5083 - va
l_loss: 1.4364
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 147s 188ms/step - accuracy: 0.4540 - loss: 1.5644 - val_accuracy: 0.5246 - va
l_loss: 1.3763
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 149s 191ms/step - accuracy: 0.4794 - loss: 1.4928 - val_accuracy: 0.5363 - va
l_loss: 1.3411
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 127s 163ms/step - accuracy: 0.4914 - loss: 1.4495 - val_accuracy: 0.5441 - va
l_loss: 1.3080
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 123s 158ms/step - accuracy: 0.5060 - loss: 1.4209 - val_accuracy: 0.5498 - va
l_loss: 1.2847
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 121s 155ms/step - accuracy: 0.5207 - loss: 1.3800 - val_accuracy: 0.5587 - va
l_loss: 1.2660
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 122s 155ms/step - accuracy: 0.5289 - loss: 1.3700 - val_accuracy: 0.5612 - va
l_loss: 1.2483
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 122s 156ms/step - accuracy: 0.5342 - loss: 1.3474 - val_accuracy: 0.5686 - va
l_loss: 1.2317
313/313 ━━━━━━━━━━━━━━━━━━━━ 25s 80ms/step - accuracy: 0.5642 - loss: 1.2305
Test accuracy: 56.86%
```

Conclusion:

Thus, we have understood the importance of Transfer learning and also the usage of transfer learning in tensorflow